

Scientific Computing Accelerated on FPGA 2022

FPGA compute acceleration with Intel[®] oneAPI

Maurizio Paolini

Field Applications Engineer, Intel Corporation

July 2022

Copyright © 2022 Intel Corporation.

This document is intended for personal use only.

Unauthorized distribution, modification, public performance,
public display, or copying of this material via any medium is strictly prohibited

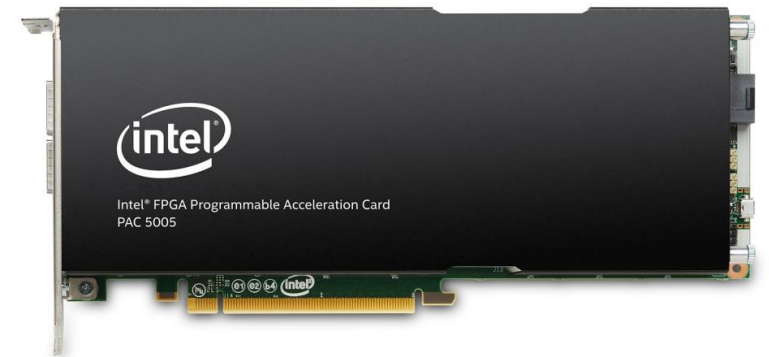


Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.

Agenda

FPGA Compute Acceleration with Intel® oneAPI

- Introduction to oneAPI
- The DPC++ programming language
- oneAPI Development Flow for FPGAs
- FPGA hardware for oneAPI
- Introduction to Intel® DevCloud



Introduction to oneAPI

Advantages of Heterogeneous Computing

Multiple Architectures

- Developers can optimize specialized inline and offload workloads to meet business needs.
 - Strengths of individual xPUs (CPU, GPU, FPGAs, etc.) can be combined for the benefit of the overall system

Performance/Watt



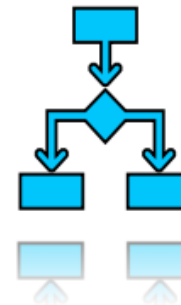
Throughput



Latency



IO Flexibility



Memory Bandwidth



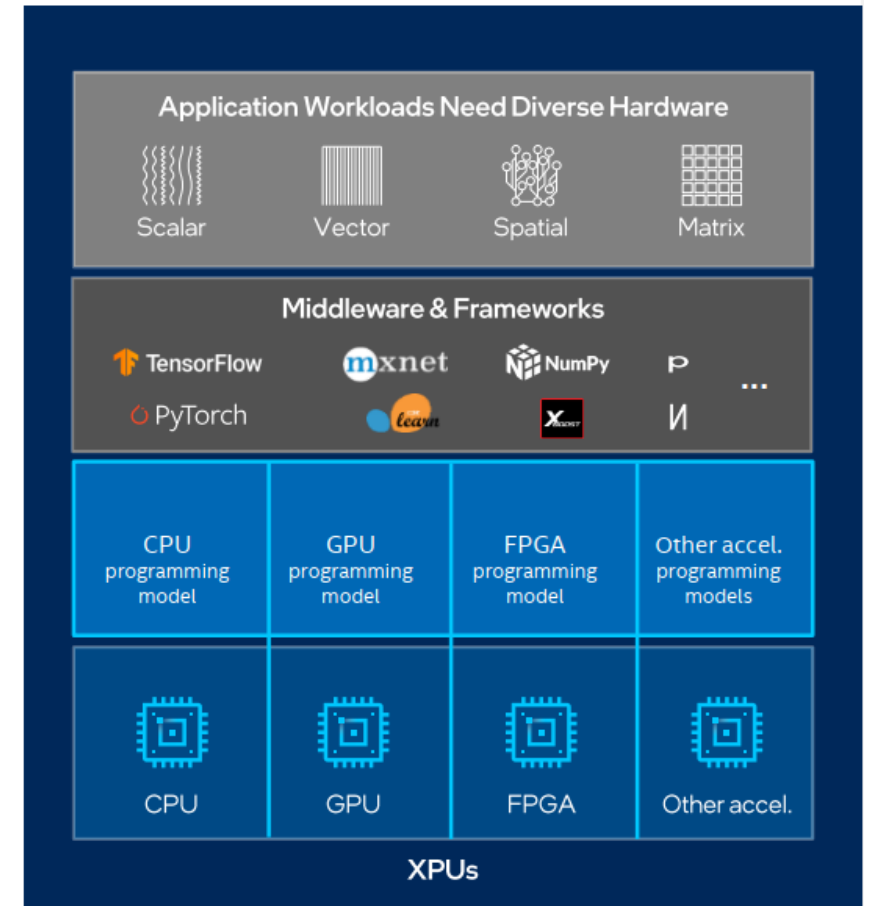
Architecture



Programming Challenges

Multiple Architectures

- **Separate** programming models and toolchains for each architecture.
 - Required **training** and **licensing** – compiler, IDE, debugger, analytics/monitoring tool, deployment tool, et al. – per architecture.
 - Challenging experience in **debug**, **monitoring**, and **maintenance** of a cross-architectural source code.
 - Difficult integration across proprietary IPs and architectures and no code re-use.
- Software development complexity **limits** freedom of architectural choice.
 - Isolated investments required for technical expertise to overcome the barrier-to-entry



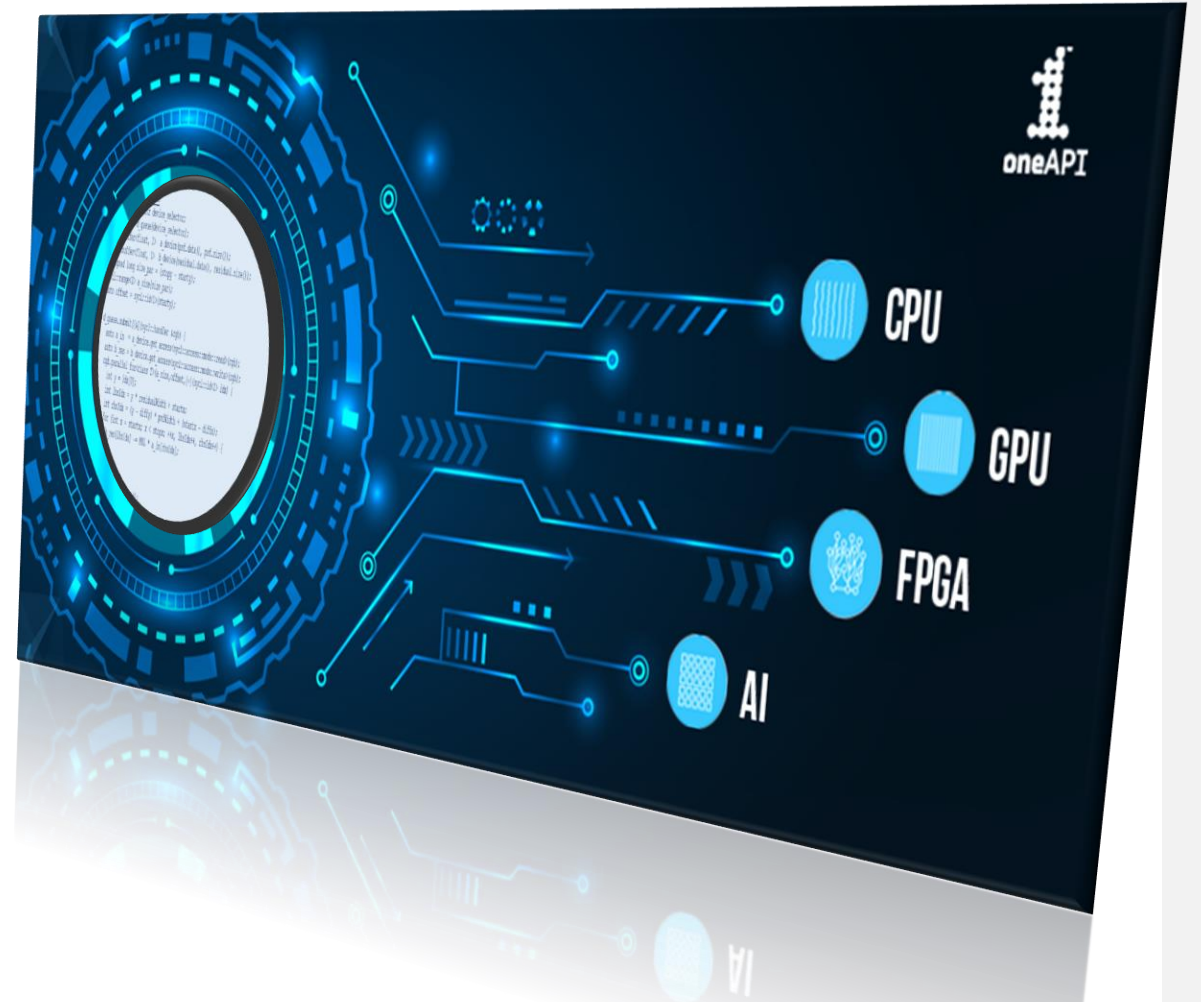
A Unified Programming Model

Multiple Architectures

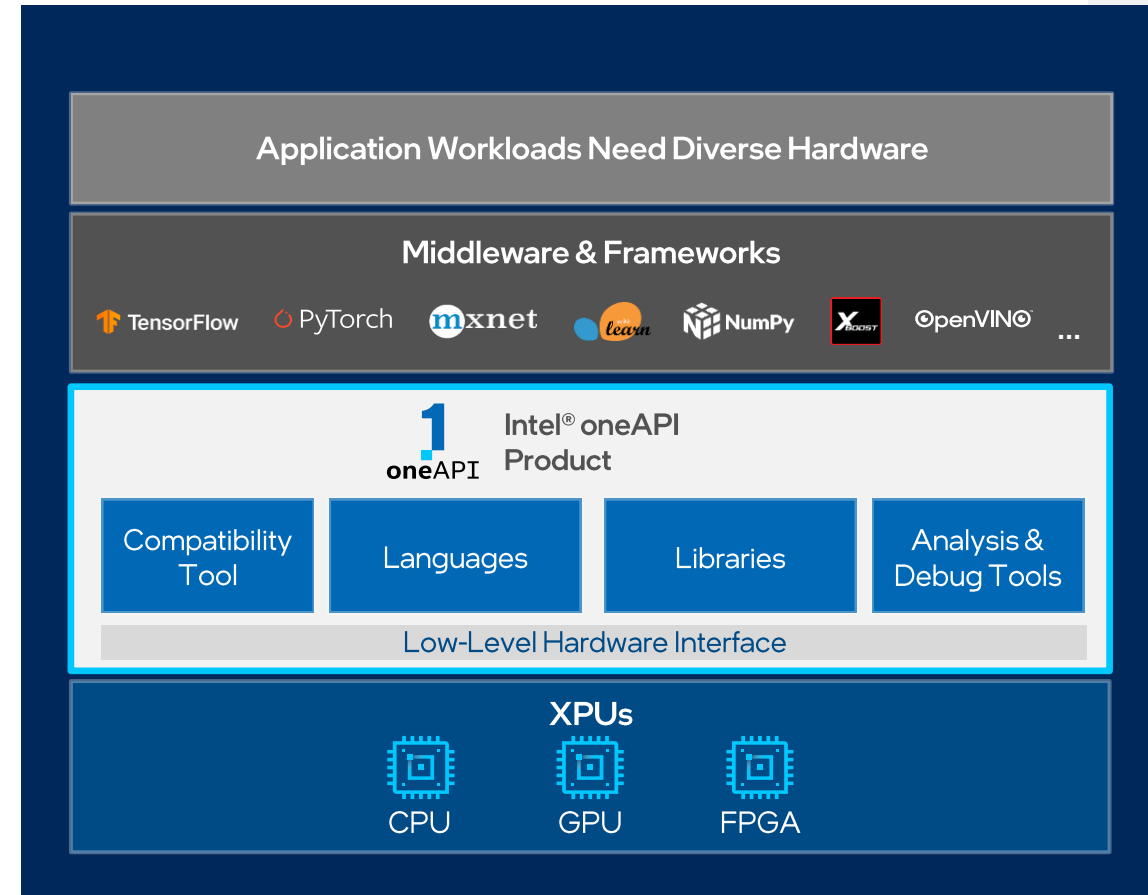
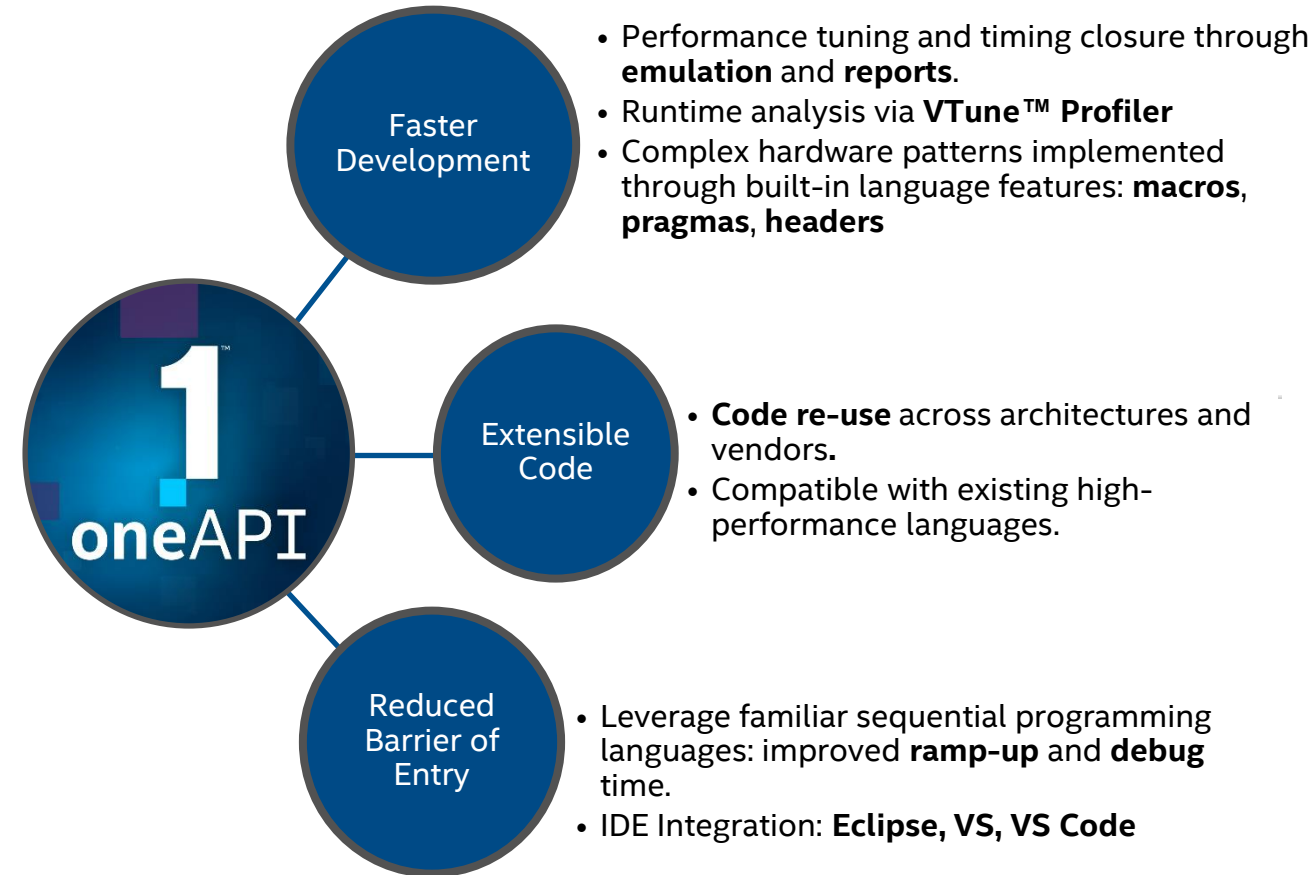
The **oneAPI** product delivers a unified programming model to simplify development across diverse architectures.

It guarantees:

- **Common developer experience** across Scalar, Vector, Matrix and Spatial architectures (CPU, GPU, AI and FPGA)
- **Uncompromised native high-level language performance**
- **Industry standardization and open specifications**

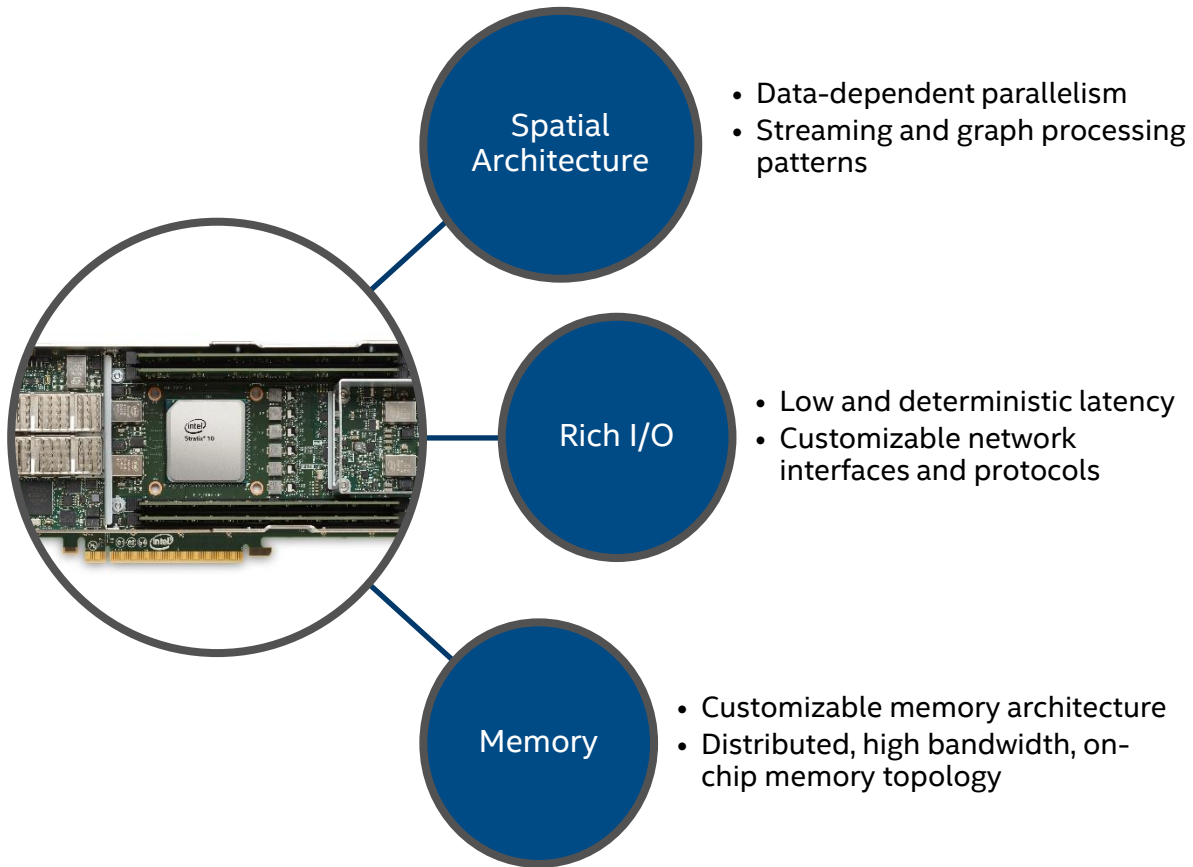


Intel® oneAPI Product

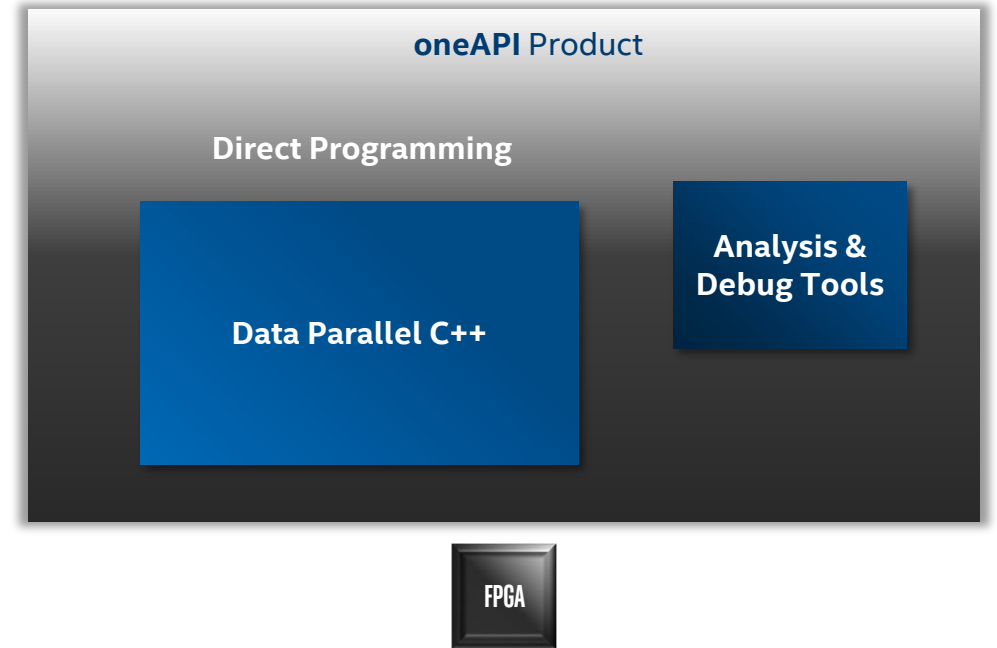


[Available Now](#)

Intel® FPGAs + Intel® oneAPI Toolkits



+



Large number of use cases

Examples

- Data compression
- Image compression
- File parsing
- Data Base acceleration
- Genomics
- Financial

Accelerating Re-Pair Compression using FPGAs

Robert Lasch, Suleyman S. Demirsoy, Norman May, Veeraraghavan Ramamurthy, Christian Färber, Kai-Uwe Sattler, Intel Corporation

ABSTRACT
Re-Pair is a compression algorithm well-suited for random accesses to compressed data. It is a highly optimized CPU version of Re-Pair that is deployed on a more resourceful FPGA for random access to compressed data. It is a highly optimized CPU version of Re-Pair that is deployed on a more resourceful FPGA for random access to compressed data. It is a highly optimized CPU version of Re-Pair that is deployed on a more resourceful FPGA for random access to compressed data.

FPGA-Accelerated Compression of Integer Vectors

Mahmoud Mohsen, Norman May, Christian Färber, David Broneske, Intel Corporation, University of Magdeburg

ABSTRACT
An efficient compression of integer vectors is critical in dictionary-based compression. An efficient compression of integer vectors is critical in dictionary-based compression. An efficient compression of integer vectors is critical in dictionary-based compression.

PipeJSON: Parsing JSON at Line Speed on FPGAs

Jonas Dann, Royden Wagner, Daniel Ritter, Christian Färber, Holger Fröning, Intel Corporation, SAP SE, Waldorf, Germany, Heidelberg University Heidelberg, Germany

Abstract
JavaScript Object Notation (JSON) is a data exchange and storage format. While modern CPUs show an improved JSON parsing performance, the limited pipelining of CPUs prevent from reaching the practical limit of performance. We present PipeJSON, the first stand-alone parser to process tens of gigabytes of JSON data on FPGAs. PipeJSON can parse multiple characters per clock cycle and achieves 7.95x speedup over JSON parsers on CPUs. Despite data size, PipeJSON achieves 7.95x speedup over JSON parsers on CPUs. Despite data size, PipeJSON achieves 7.95x speedup over JSON parsers on CPUs.

Resource-Efficient Database Query Processing on FPGAs

Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, TU Dresden & SAP SE, Intel Corporation, TU Dresden, akash.kumar@tu-dresden.de

ABSTRACT
FPGA technology has introduced new ways to accelerate database query processing, that often result in higher performance and energy efficiency. This is thanks to the unique architecture of FPGAs using reconfigurable resources to behave like an application-specific integrated circuit upon programming. The limited amount of these resources restricts the number and type of modules that an FPGA can simultaneously support. In this paper, we propose "morphing sort-merge", a set of run-time reconfigurable FPGA modules that achieve resource efficiency by reusing the FPGA's resources to support different pipeline-breaking database operators, namely sort, aggregation, and equi-join. The proposed modules use dynamic optimization mechanisms that adapt the implementation to the distribution of data at run-time, thus resulting in higher performance. Our benchmarks show that morphing sort-merge reaches an average speedup of 10x compared to MonetDB.

The DPC++ Programming Language

Data Parallel C++ (DPC++)

- Common language designed to target any XPU
 - Tuning still needed for each architecture
- Goal: to incorporate everything needed to get the best performance out of every architecture

Based on C++ and SYCL

- SYCL is based on OpenCL
- Think of it as SYCL + extensions

Allows for single-source targeting of accelerators

- Doesn't require multiple files

Open specification

DPC++: Three Scopes

- DPC++ programs consist of 3 scopes:
 - **Application scope** - Code executed on the host
 - **Command group scope** - Code for submitting data and commands to the accelerator
 - **Kernel scope** - Code executed on the accelerator
- The full capabilities of C++ are available at application and command group scope
- At kernel scope there are limitations in accepted C++
 - Most important is no recursive code
 - See SYCL specification for complete list

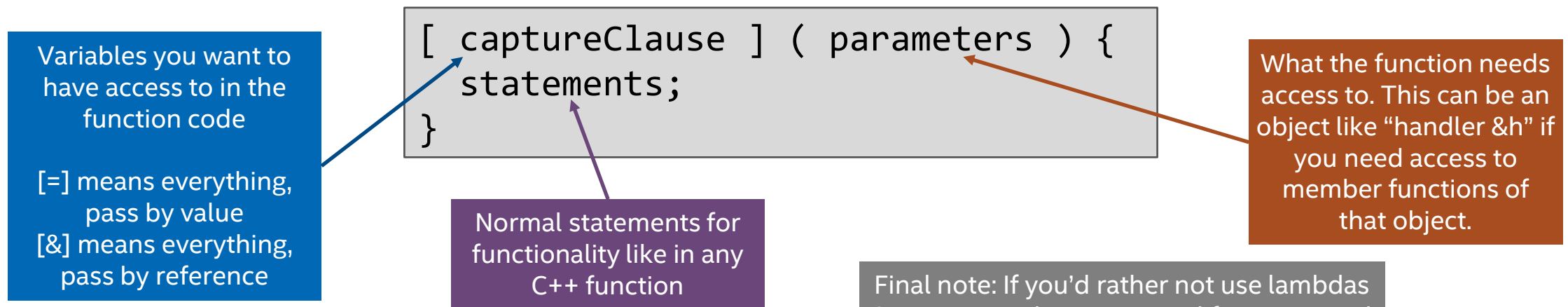
```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;           Application  
                                            Scope  
    // Set up a DPC++ device queue  
    queue q(selector);  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);    Command  
                                            Group  
                                            Scope  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];        Kernel  
                                            Scope  
            }  
        });  
    });  
}
```

The “Runtime”

- The DPC++/SYCL runtime is the program running in the background on the host controlling the execution and data passing needs of the heterogeneous compute execution
- It handles:
 - Kernel and host execution in an order imposed by data dependency needs (discussed later)
 - Passing data back and forth between the host and device
 - Querying the device
 - Etc.

A Note About Lambda Functions

- Two common constructs in DPC++ - queue submissions and kernel dispatch functions - take **function pointers** as arguments
- This doesn't lend itself to simple, in-line code
- To write simpler and neat code, **lambda functions** are used
- **Lambda functions** are un-named functions used in-line with other code
- If you are not familiar with them, here is a simple guide



DPC++ Class: `device`

- The `device` class represents the accelerators in a oneAPI system
- The `device` class contains member functions for querying information about the available devices
- The function `get_info` gives information about a device:
 - Name, vendor, and version of the device
 - Width for built in types, clock frequency, cache width and sizes, online or offline

```
// Get all of the devices a system is capable of operating
std::vector<device> my_devices = device::get_devices();
// Grab the first device to print info out for
device my_device = my_devices[0];
// Print the name of the first device
std::cout << "Device: " << my_device.get_info<info::device::name>() << std::endl;
```

DPC++ Class: `device_selector`

- The `device_selector` enables the selection of a device to execute kernels on
- Use the selector when you create a `queue` (covered next)
- The code sample shows use of several example device selectors, including an FPGA
- A custom device selector can be defined and used for targeting specific devices

```
// Other example selectors that are not FPGAs
// default_selector selector;
// host_selector selector;
// cpu_selector selector;
// gpu_selector selector;

// Create the selector as an fpga_selector type
INTEL::fpga_selector selector;

// Use the selector when you create a queue
queue q(selector);
```


DPC++ Class: `queue`

- A `queue` is a mechanism where work is submitted to a `device`
- A queue submits command groups to be executed by the SYCL runtime
- A `queue.submit()` is the beginning of the command group scope
 - Groups of work to be executed by the SYCL runtime on an accelerator
- A queue maps to a single device

```
// Declare a queue to a device
queue q(selector);

// Submit things to the queue
q.submit([&](handler& h) {
    // COMMAND GROUP CODE
});
```

The handler is a class that contains all of the command group functions of SYCL

You can think of it as an abstraction of the runtime

This keeps us from having to type `handler::` again and again in the command group scope

DPC++ Class: `kernel`

- The `kernel` encapsulates code that will be run on the accelerator
- A `kernel` object is not explicitly constructed by the user
- It is constructed when a kernel dispatch function, such as `parallel_for()` or `single_task()` is called

```
q.submit([&](handler& h) {  
  
    // The "kernel" is everything after the kernel dispatch function  
    h.single_task<VectorAdd>([=]() {  
  
        // Everything inside here is "KERNEL SCOPE"  
        for (int i = 0; i < kSize; ++i) {  
            c[i] = a[i] + b[i];  
        }  
    });
```

Single Task Kernels

- `single_task()` kernels allow complex or lengthy datapaths to be built from custom hardware in FPGAs
- Useful to offload code with dependencies that are difficult to execute in a data parallel fashion
- Look like CPU code
 - Contain an outer loop to process all data
- Ideal for & recommended for **FPGAs**

```
for(int i=0; i < 1024; i++){  
    a[i] = b[i] + c[i];  
});
```

**CPU
Implementation**

```
h.single_task([=](){  
    for (int i=0; i < 1024; i++) {  
        A[i] = B[i] + C[i];  
    }  
});
```

**FPGA Kernel
Implementation**

DPC++ Class: `buffer` and `accessor`

■ `buffer`

- Encapsulates data in a SYCL application
- Across both devices and host!

■ `accessor`

- Mechanism to access buffer data
- Determines data dependencies in that order kernel executions (covered later)

```
int main() {
    ... // Code to set up standard C++ vectors

    buffer buf_a(vector_a);
    buffer buf_b(vector_b);
    buffer buf_c(vector_c);

    queue q(selector);

    q.submit([&](handler& h) {
        accessor a(buf_a, h, read_only);
        accessor b(buf_b, h, read_only);
        accessor c(buf_c, h, write_only);

        h.single_task<VectorAdd>([=]() {
            for (int i = 0; i < kSize; i++) {
                c[i] = a[i] + b[i];
            }
        });
    });
};
```

#Include Files

- oneAPI programs require the include of `cl/sycl.hpp`
- Programs targeting FPGAs require the include of `cl/sycl/INTEL/fpga_extensions.hpp`

```
// Always include these at the top of your program
```

```
#include <CL/sycl.hpp>
```

```
#include <CL/sycl/INTEL/fpga_extensions.hpp>
```

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
    // Set up a DPC++ device queue  
    queue q(selector);  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c, int N) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command group for execution

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < kSize; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < N; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data

Step 6: Send a kernel for execution

DPC++ Simple Program Walk-Through

```
void dpcpp_code(int* a, int* b, int* c) {  
    //Set up an FPGA device selector  
    INTEL::fpga_selector selector;  
  
    // Set up a DPC++ device queue  
    queue q(selector);  
  
    // Setup buffers for input and output vectors  
    buffer buf_a(a, range<1>(N));  
    buffer buf_b(b, range<1>(N));  
    buffer buf_c(c, range<1>(N));  
  
    //Submit Command group function object to the queue  
    q.submit([&](handler &h){  
  
        //Create device accessors to buffers  
        accessor a(buf_a, h, read_only);  
        accessor b(buf_b, h, read_only);  
        accessor c(buf_c, h, write_only);  
  
        //Dispatch the kernel  
        h.single_task<VectorAdd>([=]() {  
            for (int i = 0; i < N; i++) {  
                c[i] = a[i] + b[i];  
            }  
        });  
    });  
}
```

Step 1: Create a device selector targeting the FPGA

Step 2: Create a device queue, using the FPGA device selector

Step 3: Create buffers

Step 4: Submit a command for execution

Step 5: Create buffer accessors so the FPGA can access the data

Step 6: Send a kernel for execution

Done!

The contents of buf_c are copied to *c when the function finishes

(because of the buffer destruction of buf_c)

```

int main() {
  auto R = range<1>{ num };
  buffer<int> A{ R }, B{ R };
  queue Q;

  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); }); } Kernel 1

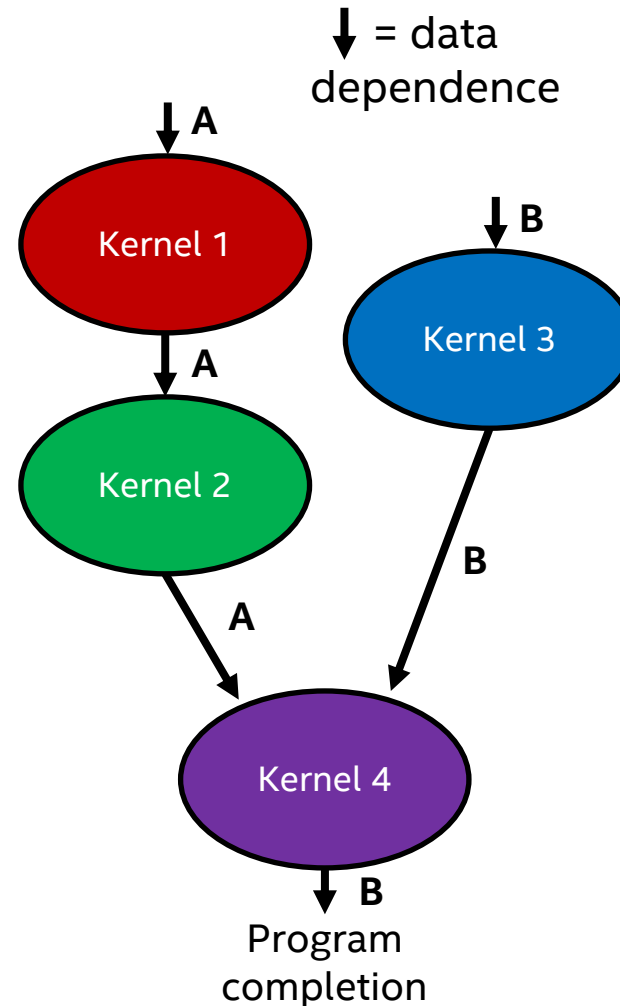
  Q.submit([&](handler& h) {
    auto out = A.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); }); } Kernel 2

  Q.submit([&](handler& h) {
    auto out = B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      out[idx] = idx[0]; }); }); } Kernel 3

  Q.submit([&](handler& h) {
    auto in = A.get_access<access::mode::read>(h);
    auto inout =
      B.get_access<access::mode::read_write>(h);
    h.parallel_for(R, [=](id<1> idx) {
      inout[idx] *= in[idx]; }); }); } Kernel 4
}

```

Kernel Execution Order

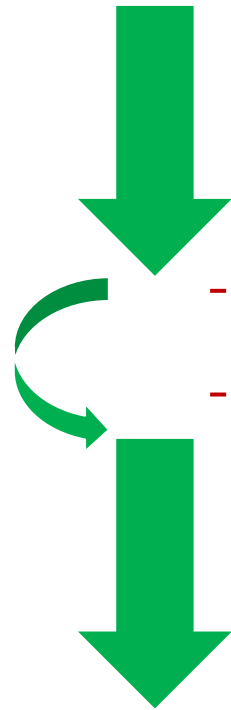


- Kernels can execute at the same time
 - If no data dependences
- Accessors are used to determine dependences
- Execution ordering is automatically determined

Asynchronous Host/Kernel Execution

- The execution of the host code is asynchronous to what is being executed on the accelerator
- If you need synchronization, you must impose that yourself

Host code execution



```
#include <CL/sycl.hpp>
#include <iostream>
constexpr int num=16;
using namespace cl::sycl;

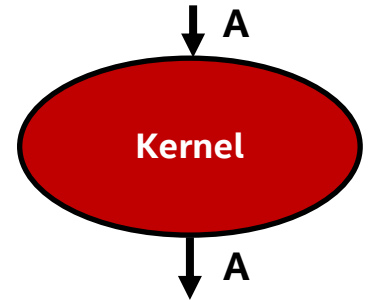
int main() {
    auto R = range<1>{ num };
    buffer<int> A{ R };

    queue{}.submit([&](handler& h) {
        auto out =
A.get_access<access::mode::write>(h);
        h.parallel_for(R, [=](id<1> idx) {
            out[idx] = idx[0]; });
    });

    auto result =
A.get_access<access::mode::read>();
    for (int i=0; i<num; ++i)
        std::cout << result[i] << "\n";

    return 0;
}
```

Kernel Execution



Synchronization Method 1: Host Accessor

- In the **command scope**, accessors are created for the accelerator

- In the **application scope**, accessors are created for the host

- A host accessor creates a dependency node in the execution graph
 - Execution at the host is blocked until the data is ready

```
int main() {  
    constexpr int N = 100;  
    auto R = range<1>(N);  
    std::vector<double> v(N, 10);  
    queue q;
```

```
    buffer<double, 1> buf(v.data(), R);  
    q.submit([&](handler& h) {  
        auto a = buf.get_access<access::mode::read_write>(h);  
        h.parallel_for(R, [=](id<1> i) {  
            a[i] -= 2;  
        });  
    });
```

Command Scope

```
    auto b = buf.get_access<access::mode::read>();  
    for (int i = 0; i < N; i++)  
        std::cout << v[i] << "\n";  
    return 0;
```

Application Scope

```
}
```

Synchronization Method 2: Buffer Destruction

- Buffer creation happens within a separate function scope
- When execution advances beyond this function scope, buffer destructor is invoked
- Relinquishes ownership of data and copies back the data to the host memory
- Scope can also be created with simple use of { }

```
#include <CL/sycl.hpp>
constexpr int N=100;
using namespace cl::sycl;

void dpcpp_code(std::vector<double> &v, queue &q){
    auto R = range<1>(N);
    buffer<double, 1> buf(v.data(), R);
    q.submit([&](handler& h) {
        auto a = buf.get_access<access::mode::read_write>(h);
        h.parallel_for(R, [=](id<1> i) {
            a[i] -= 2;
        });
    });
}

int main() {
    std::vector<double> v(N, 10);
    queue q;
    dpcpp_code(v,q);
    for (int i = 0; i < N; i++)
        std::cout << v[i] << "\n";
    return 0;
}
```


Unified Shared Memory

- USM => unified virtual address space
 - Any pointer value returned by a USM allocation routine is valid on the device
 - Three different types of allocations are defined
 - Device - data on device, must be explicitly transferred, not accessible from host
 - Host - data on host, accessible from device
 - Shared - data on host and/or on device, runtime managed data movement
 - Host allocation ideal for streaming applications at interface speed

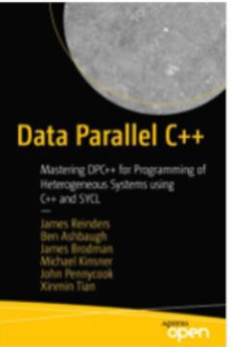
```
562 | q.submit([&](handler& h) {
563 |     h.single_task<parallelAddKernel>([=]() [[intel::kernel_args_restrict]] {
564 |
565 |         host_ptr<int> in(in_host);
566 |         host_ptr<int> out(out_host);
567 |
568 |         int i_cnt = 0;
569 |
570 |         // Init regs to zero
571 |         [[intel::fpga_register]] std::array<int, 16> raw_data;
572 |         #pragma unroll
573 |         for (int i = 0; i < 16; i++) { raw_data[i] = 0; }
574 |
575 |         [[intel::fpga_register]] std::array<int, 8> add_data;
576 |         #pragma unroll
577 |         for (int i = 0; i < 8; i++) { add_data[i] = 0; }
578 |
579 |
580 |
581 |         // Run over all CLs
582 |         while(i_cnt < iterations) {
583 |
584 |             // Load complete CL in one clock cycle, (same for PCIe and DDR4)
585 |             #pragma unroll
586 |             for (uint idx = 0; idx < 16; idx++) {
587 |                 {
588 |                     raw_data[idx] = in[idx + i_cnt*16];
589 |                 }
590 |
591 |
592 |             add_data[0] = raw_data[0] + raw_data[8];
593 |             add_data[1] = raw_data[1] + raw_data[9];
594 |             add_data[2] = raw_data[2] + raw_data[10];
595 |             add_data[3] = raw_data[3] + raw_data[11];
596 |             add_data[4] = raw_data[4] + raw_data[12];
597 |             add_data[5] = raw_data[5] + raw_data[13];
598 |             add_data[6] = raw_data[6] + raw_data[14];
599 |             add_data[7] = raw_data[7] + raw_data[15];
600 |
601 |
602 |             // Write results back with half CL, as we can write and read
603 |             // a CL per clock cycle this will create no bottleneck
604 |             #pragma unroll
605 |             for (uint idx = 0; idx < 8; idx++) {
606 |                 {
607 |                     out[idx + i_cnt*8] = add_data[idx];
608 |                 }
609 |
610 |             }
611 |
612 |             i_cnt++;
613 |
614 |         }
615 |     });
616 | }
```

Learn More About DPC++

- Download DPC++ book for free
 - <https://link.springer.com/book/10.1007%2F978-1-4842-5574-2>

- DPC++ Training Modules

- https://devcloud.intel.com/oneapi/get_started/baseTrainingModules/



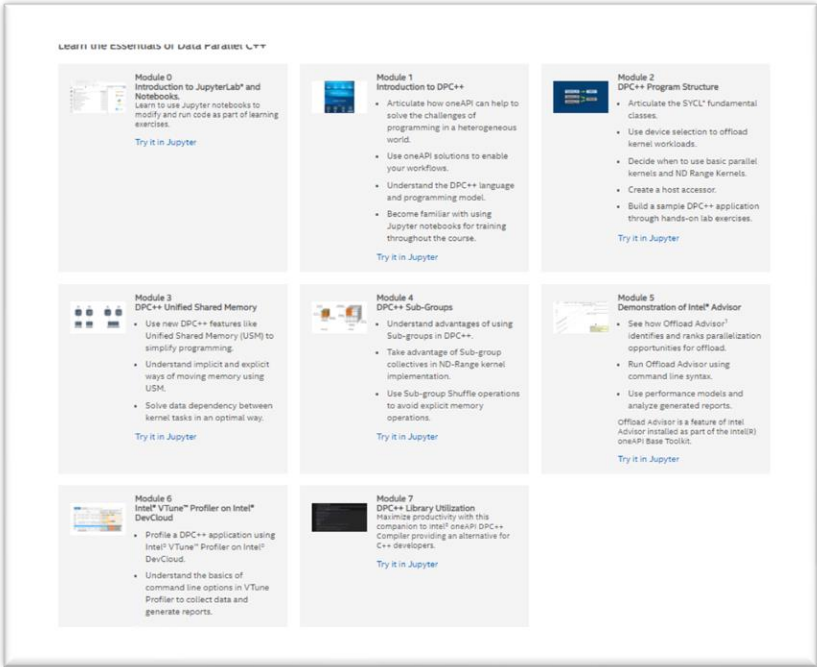
Data Parallel C++
Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL

Authors [\(view affiliations\)](#)
James Reinders, Ben Ashbaugh, James Brodman, Michael Kinsner, John Pennycook, Xinmin Tian

Open Access | Book

1 Citations | 21 Mentions | 207k Downloads

[Download book PDF](#) [Download book EPUB](#)



LEARN THE ESSENTIALS OF DATA PARALLEL C++

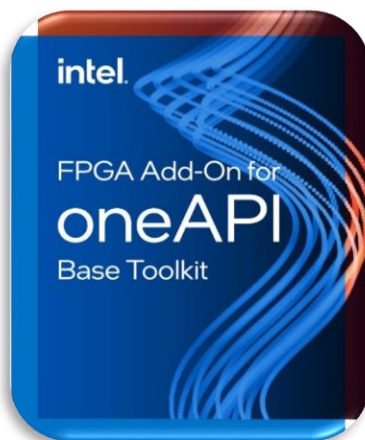
- Module 0** Introduction to JupyterLab® and Notebooks.
Learn to use Jupyter notebooks to modify and run code as part of learning exercises.
Try it in Jupyter
- Module 1** Introduction to DPC++
• Articulate how oneAPI can help to solve the challenges of programming in a heterogeneous world.
• Use oneAPI solutions to enable your workflows.
• Understand the DPC++ language and programming model.
• Become familiar with using Jupyter notebooks for training throughout the course.
Try it in Jupyter
- Module 2** DPC++ Program Structure
• Articulate the SYCL® fundamental classes.
• Use device selection to offload kernel workloads.
• Decide when to use basic parallel kernels and ND Range Kernels.
• Create a host accessor.
• Build a sample DPC++ application through hands-on lab exercises.
Try it in Jupyter
- Module 3** DPC++ Unified Shared Memory
• Use new DPC++ features like Unified Shared Memory (USM) to simplify programming.
• Understand implicit and explicit ways of moving memory using USM.
• Solve data dependency between kernel tasks in an optimal way.
Try it in Jupyter
- Module 4** DPC++ Sub-Groups
• Understand advantages of using Sub-groups in DPC++.
• Take advantage of Sub-group collectives in ND-Range kernel implementation.
• Use Sub-group Shuffle operations to avoid explicit memory operations.
Try it in Jupyter
- Module 5** Demonstration of Intel® Advisor
• See how Offload Advisor¹ identifies and ranks parallelization opportunities for offload.
• Run Offload Advisor using command line syntax.
• Use performance models and analyze generated reports.
Offload Advisor is a feature of Intel Advisor installed as part of the Intel® oneAPI Base Toolkit.
Try it in Jupyter
- Module 6** Intel® VTune™ Profiler on Intel® DevCloud
• Profile a DPC++ application using Intel® VTune™ Profiler on Intel® DevCloud.
• Understand the basics of command line options in VTune Profiler to collect data and generate reports.
- Module 7** DPC++ Library Utilization
Maximize productivity with this companion to Intel® oneAPI DPC++ Compiler providing an alternative for C++ developers.
Try it in Jupyter

OneAPI Development Flow for FPGAs

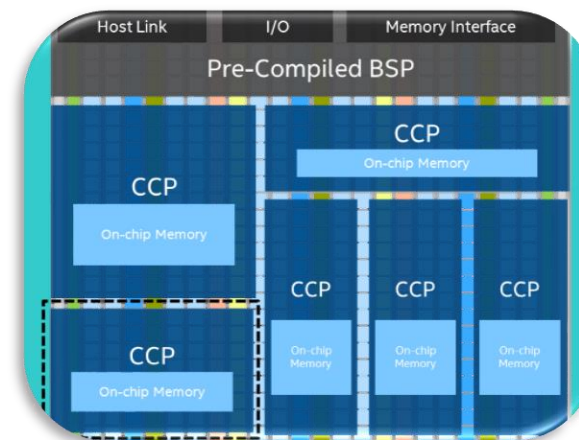
Getting Started with oneAPI on an FPGA



Intel® oneAPI Base Toolkit



Intel® FPGA Add-on for oneAPI Base Toolkit



Board Support Package (BSP)

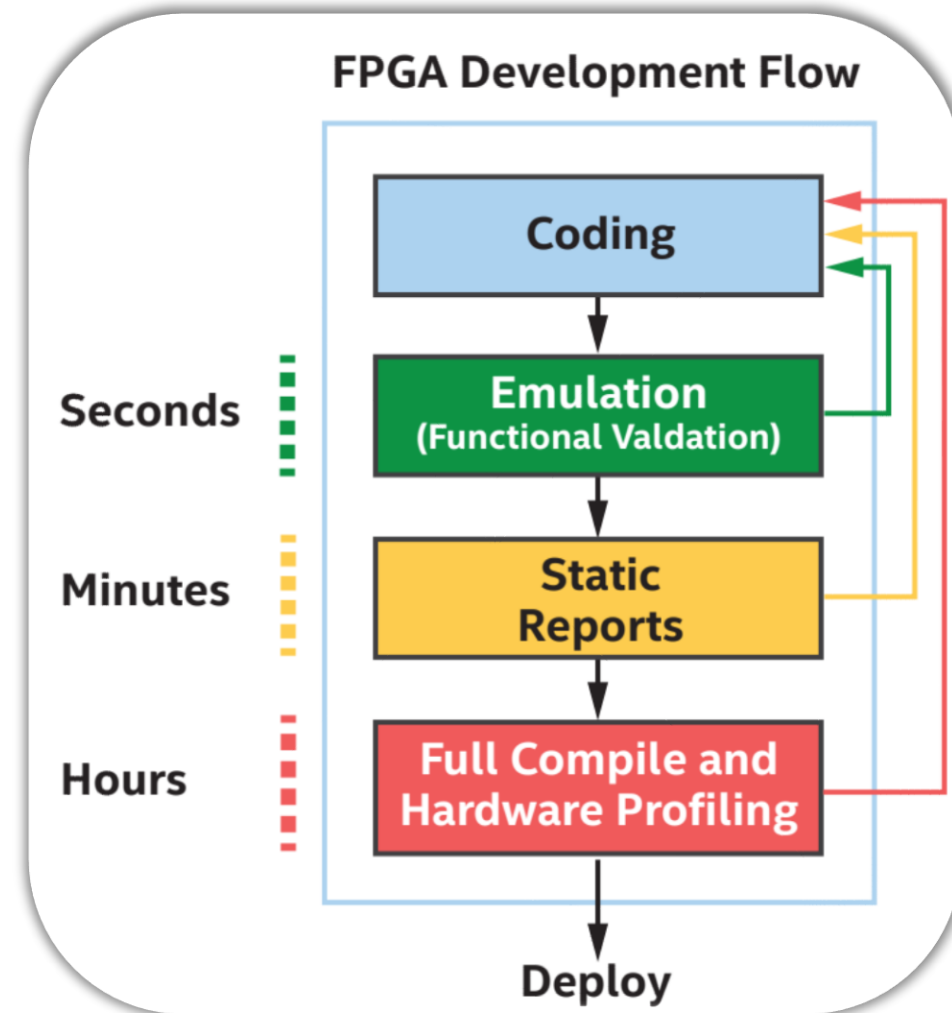
Note: Developers using custom platforms should [download](#) the Intel® FPGA Add-on for Intel® Custom Platforms with the respective Intel® Quartus® version and obtain a BSP from their 3rd part platform vendor.

Installing oneAPI

- Get started by visiting the Intel[®] Software Developer Zone landing page for the Intel[®] oneAPI Toolkits
 - <https://software.intel.com/en-us/oneapi>
- Get the Intel[®] oneAPI Base Toolkit for Linux*
 - Supports compiles for emulation and the optimization report
- Install the Intel[®] FPGA Add-on for oneAPI Base Toolkit
 - Needed for compiles to FPGA hardware
 - Contains Intel[®] Quartus[®] Prime software “under the hood,” be sure to comply to required versions of operating system

FPGA Development Flow for oneAPI Projects

- FPGA Emulator target (Emulation)
 - Compiles in seconds
 - Runs completely on the host
- Optimization report generation
 - Compiles in seconds to minutes
 - Identify bottlenecks
- FPGA bitstream compilation
 - Compiles in hours
 - Enable profiler to get runtime analysis



Anatomy of a dpcpp Command Targeting FPGAs

```
dpcpp -fintel fpga *.cpp/*.o [device link options] [-Xs arguments]
```

Target Platform

Link Options

FPGA-Specific Arguments

Language
DPCPP = Data
Parallel C++

Input Files
source or object

Emulation

Seconds of Compilation

Does my code give me the correct answers?

Quickly generate code that runs on the x86 host to emulate the FPGA

Developers can:

- Verify functionality of design through CPU compile and emulation.
- Identify quickly syntax and pointer implementation errors for iterative design/algorithm development.
- Enable deep, system-wide debug with Intel[®] Distribution for GDB.
- Functional debug of SYCL code with FPGA extensions.

Emulation Command

```
#ifdef FPGA_EMULATOR
    intel::fpga_emulator_selector device_selector;
#else
    intel::fpga_selector device_selector;
#endif
```

Include this construct in
your code

```
dpcpp -fintel_fpga <source_file>.cpp -DFPGA_EMULATOR
```



Report Generation

Minutes of Compilation

Where are the bottlenecks?

Quickly generate a report to guide optimization efforts

Developers can:

- Identify any memory, performance, data-flow bottlenecks in their design.
- Receive suggestions for optimization techniques to resolve said bottlenecks.
- Get area and timing estimates of their designs for the desired FPGA.

Command to Produce an Optimization Report

Two Step Method:

```
dpcpp -fintel FPGA <source_file>.cpp -c -o <file_name>.o  
dpcpp -fintel FPGA <file_name>.o -fsycl-link -Xshardware
```

One Step Method:

```
dpcpp -fintel FPGA <source_file>.cpp -fsycl-link -Xshardware
```

The default value for `-fsycl-link` is `-fsycl-link=early` which produces an early image object file and report

- A report showing optimization, area, and architectural information will be produced in `<file_name>.prj/reports/`

Bitstream Compilation



Runs Intel Quartus Prime Software “under the hood” (no licensing required)

Developers can:

- Compile FPGA bitstream for their design and run it on an FPGA.
- Attain automated timing closure.
- Obtain in-hardware verification.
- Take advantage of Intel[®] VTune[™] Profiler for real-time analysis of design.

Compile to FPGA Executable with Profiler

Two Step Method:

```
dpcpp -fintel fpga <source_file>.cpp -c -o <file_name>.o  
dpcpp -fintel fpga <file_name>.o -Xshardware -Xsprofile
```

One Step Method:

```
dpcpp -fintel fpga <source_file>.cpp -Xshardware -Xsprofile
```

The profiler will be instrumented within the image and you will be able to run the executable to return information to import into Intel® Vtune Amplifier.

To compile to FPGA executable without profiler, leave off `-Xsprofile`.

Compiling FPGA Device Separately and Linking

- In the default case, the DPC++ Compiler handles generating the host executable, device image, and final executable
- It is sometimes desirable to compile the host and device separately so changes in the host code do not trigger a long compile

Partition code

has_kernel.cpp

host_only.cpp

Then run this command to compile the FPGA image:

```
dpcpp -fintelfpga has_kernel.cpp -fsycl-link=image -o has_kernel.a -Xhardware
```

This command to produce an object file out of the host only code:

```
dpcpp -fintelfpga host_only.cpp -c -o host_only.o
```

This command to put the object files together into an executable:

```
dpcpp -fintelfpga has_kernel.a host_only.o -o executable.fpga
```

This is the long compile

Porting OpenCL code to oneAPI

- Same programming flow => easy porting
- Migration guidelines available at <https://www.intel.com/content/www/us/en/develop/documentation/migrate-opencl-fpga-designs-to-dpcpp/top.html>



Kernel uses C99	Kernel uses DPC++
Autorun kernel	No autorun kernel but easy workaround available
Buffer read/write access is from the host point of view	Buffer access is from the device point of view
Buffers with clEnqueueWriteBuffer or USM buffers	Buffers and accessors controlled by SYCL runtime or USM pointers
Programming file AOCX separated from host executable	Single executable combining FPGA programming file and host executable – fat binary
VTune profiling only from host system	VTune profiling from host and kernel system

Intel® VTune™

Start

- Add `-Xsprofile` to the set of flags in Makefile of hardware compile
- Start VTune with root rights and configure analysis CPU/FPGA

Configure Analysis [🔗](#)

WHERE

Local Host ▾

WHAT

Launch Application ▾

Specify and configure your analysis target: an application or a script to execute.

Application:

Application parameters:

Use application directory as working directory

Advanced >

HOW

CPU/FPGA Interaction ▾

Analyze CPU/FPGA interaction issues through these ways: 1. Focus on the kernels running on the FPGA. 2. Identify the most time-consuming kernels. 3. Look at the corresponding metrics on the device side (like Occupancy or Stalls). 4. Correlate with CPU and platform profiling data. [Learn more](#)

CPU sampling interval, ms

Collect stacks

FPGA profiling data source

Path to .aocx or host binary file

FPGA readback period

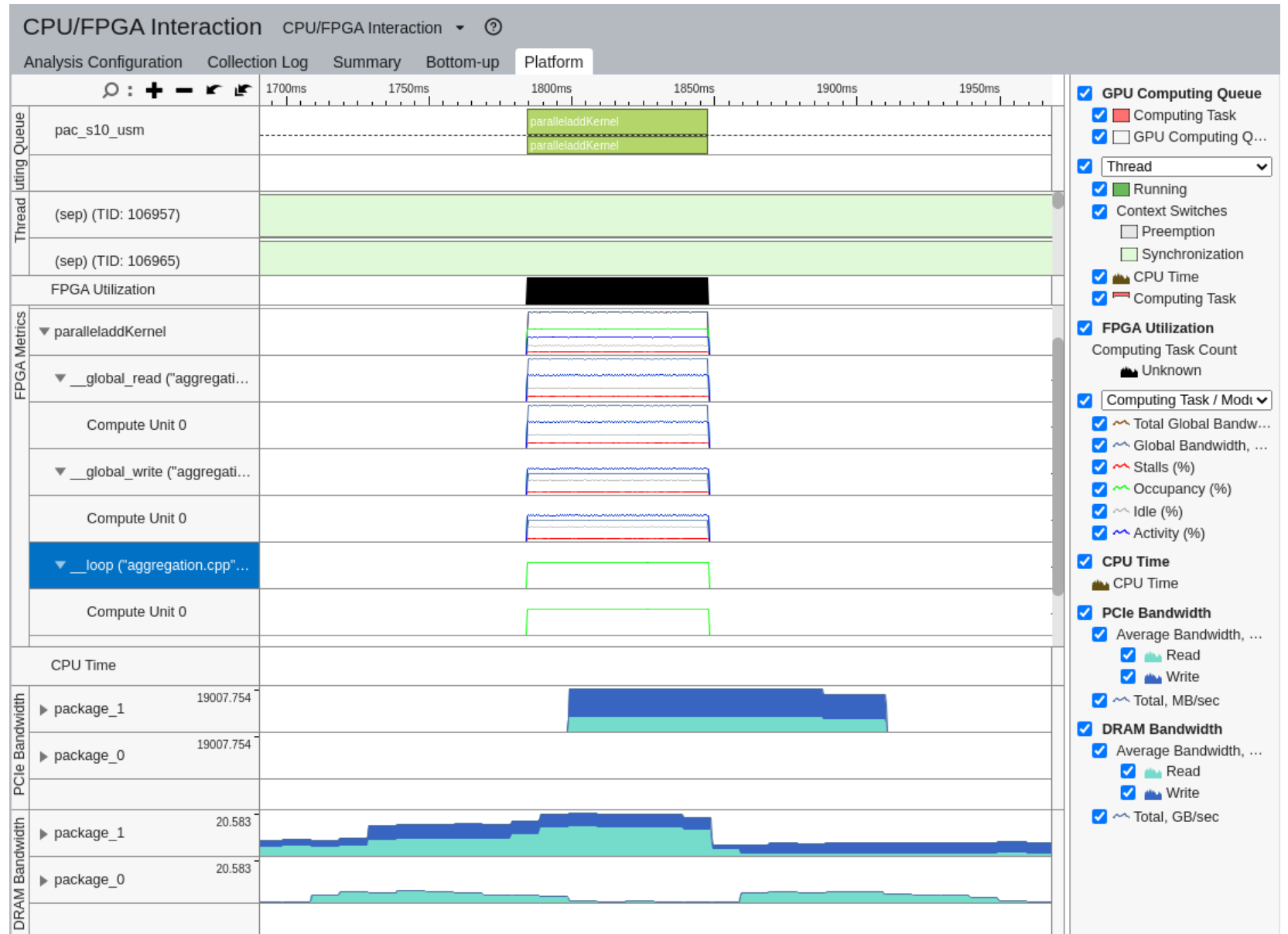
FPGA no temporal

FPGA no memory transfers

Details >

Intel® VTune™

- Kernel schedule in respect to host
- FPGA utilization
- PCIe BW
- DRAM BW



Intel® VTune™

- Performance counters inside kernel
- Bottleneck inside kernel visible
- Utilization of resources and throughput of each loop in kernel

CPU/FPGA Interaction		Device Metrics					
Source Line ▲	Source	Stalls (%)	Occupancy (%)	Idle (%)	Activity (%)	Data Transferred	
						Data Transfer Size	Average Bandwidth, GB/s
693	q.submit([&](handler& h) {						
694	h.single_task<parallelAddKernel>(&=)() [[intel::kernel_args_restrict]] {						
695							
696	host_ptr<int> in(in_host);						
697	host_ptr<int> out(out_host);						
698							
699	int i_cnt = 0;						
700							
701	// Init regs to zero						
702	[[intel::fpga_register]] std::array<int, 16> raw_data;						
703	#pragma unroll						
704	for (int i = 0; i < 16; i++) { raw_data[i] = 0; }						
705							
706	[[intel::fpga_register]] std::array<int, 8> add_data;						
707	#pragma unroll						
708	for (int i = 0; i < 8; i++) { add_data[i] = 0; }						
709							
710							
711							
712	// Run over all CLs						
713	while(i_cnt < iterations) {	0.0%	60.4%	0.0%	0.0%	0 B	0.000
714							
715	// Load complete CL in one clock cycle, (same for PCIe and DDR4)						
716	#pragma unroll						
717	for (uint idx = 0; idx < 16; idx++) {						
718	{						
719	raw_data[idx] = in[idx + i_cnt*16];	13.5%	61.8%	31.9%	61.8%	799.9 MB	12.647
720	}						
721	}						
722							
723	add_data[0] = raw_data[0] + raw_data[8];						
724	add_data[1] = raw_data[1] + raw_data[9];						
725	add_data[2] = raw_data[2] + raw_data[10];						
726	add_data[3] = raw_data[3] + raw_data[11];						
727	add_data[4] = raw_data[4] + raw_data[12];						
728	add_data[5] = raw_data[5] + raw_data[13];						
729	add_data[6] = raw_data[6] + raw_data[14];						
730	add_data[7] = raw_data[7] + raw_data[15];						
731							
732	//Write results back with half CL, as we can write and read a CL p						
733	#pragma unroll						
734	for (uint idx = 0; idx < 8; idx++) {						
735	{						
736	out[idx + i_cnt*8] = add_data[idx];	6.8%	61.8%	35.1%	61.8%	400.2 MB	6.327
737	}						
738	}						
739	}						

Use of RTL Libraries for FPGA in oneAPI

- Create a static library file using RTL

- `fpga_crossgen: rtl -> object`
- `fpga_libtool: objects -> library`

Files needed:

- RTL wrapper
- XML description
- Emulation model file (SYCL-based)

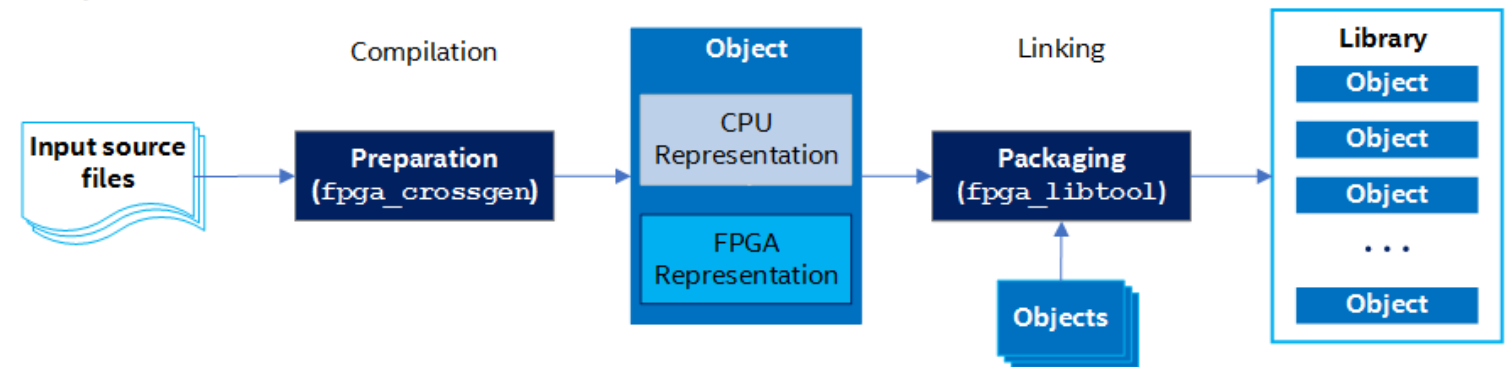
RTL design constraints:

- RTL module must have a clock port, a resetn port, and Avalon® streaming interface input and output ports
- A single pair of ready and valid logic must control all the inputs
- Declare the RTL module as stall-free if possible

- Include library file to use the functions inside your SYCL* kernels.

```
dpcpp -fintelfpga main.cpp lib.a
```

Library Toolchain Creation Process



FPGA Hardware for oneAPI

FPGA Boards and Board Support Packages

To interface with computers, FPGAs must be mounted onto boards

Boards provide:

- power and thermal management

- memory

- physical interfaces between the FPGA and other devices

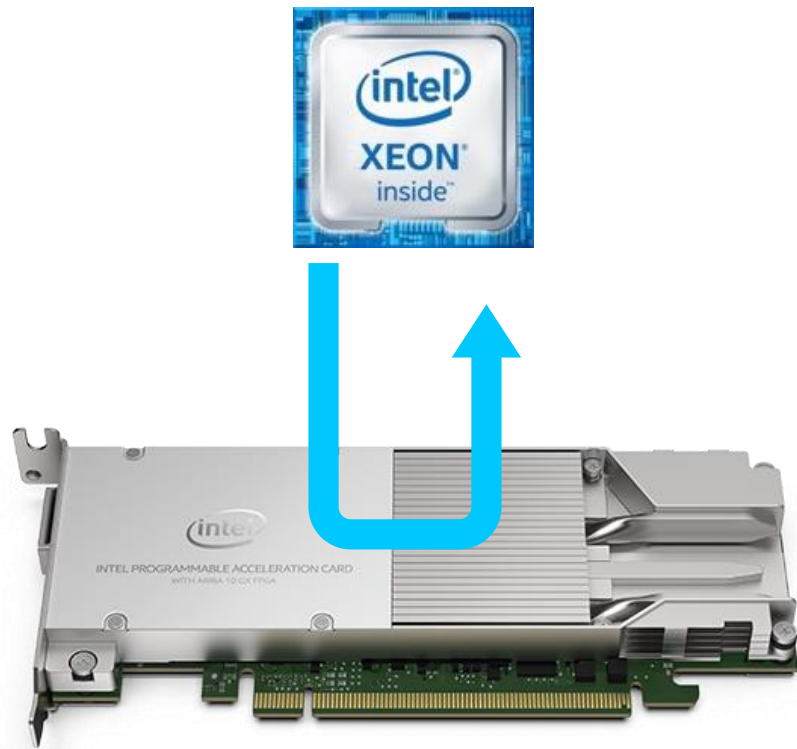
Board Support Packages (BSPs) => interface between boards and the Intel oneAPI compiler

A BSP consists of:

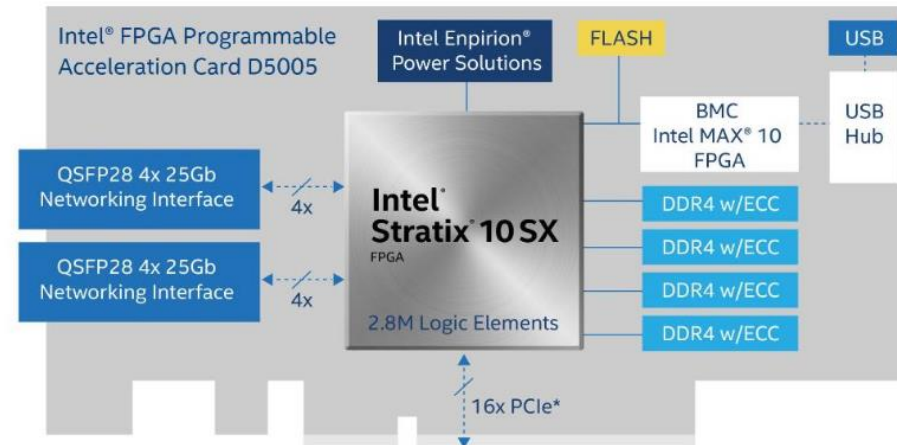
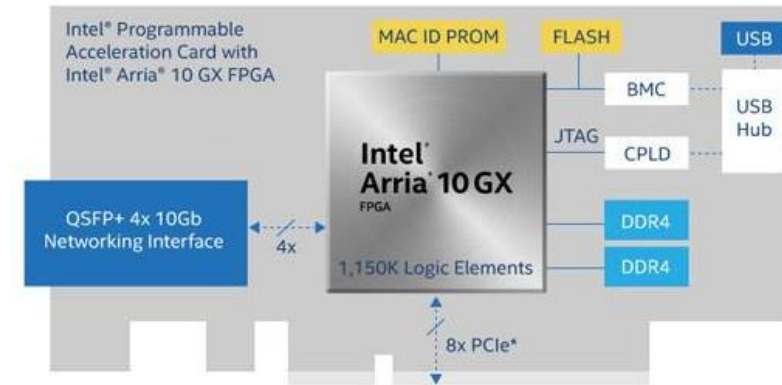
- software layers (card drivers, card management code)

- a precompiled FPGA hardware design implementing memory and physical interfaces and card management logic

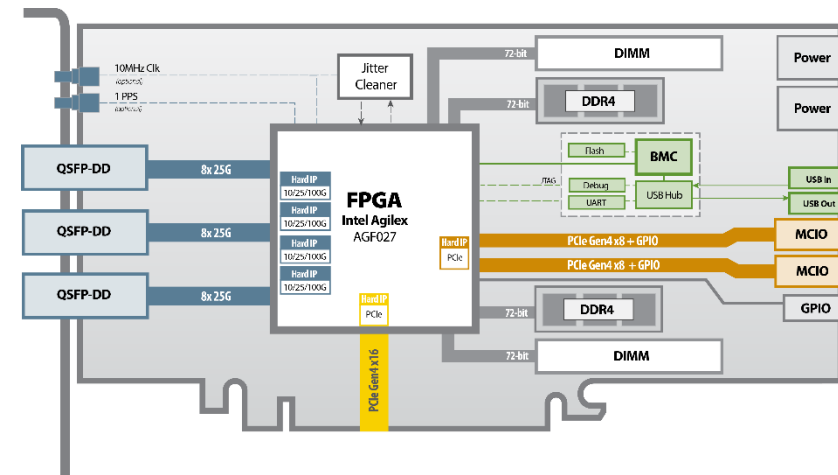
The FPGA design implementing the kernel(s) is stitched by the compiler into the framework provided by the BSP



Intel® FPGA Cards Available for use with oneAPI



Example of Intel® Agilex® FPGA Card with oneAPI



Bittware®: IA-840F

FPGA supported Technologies

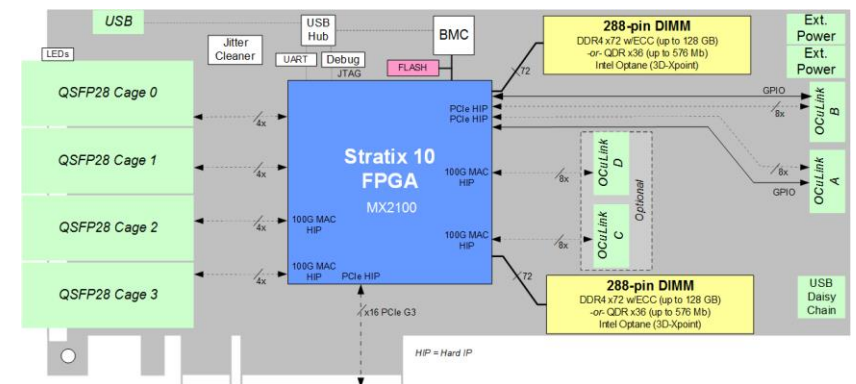
FPGA Advantages

- UPI (coming soon: CXL)
 - Cache-coherent, high bandwidth and low latency interface
 - Memory extension with DDR4/DDR-T
- High Bandwidth Memory
 - In package 16GB HBM2, up to 512GB/s
- IKL – FPGA to FPGA connection
- Network interfaces

Intel® S10DX



Bittware®: 520N-MX



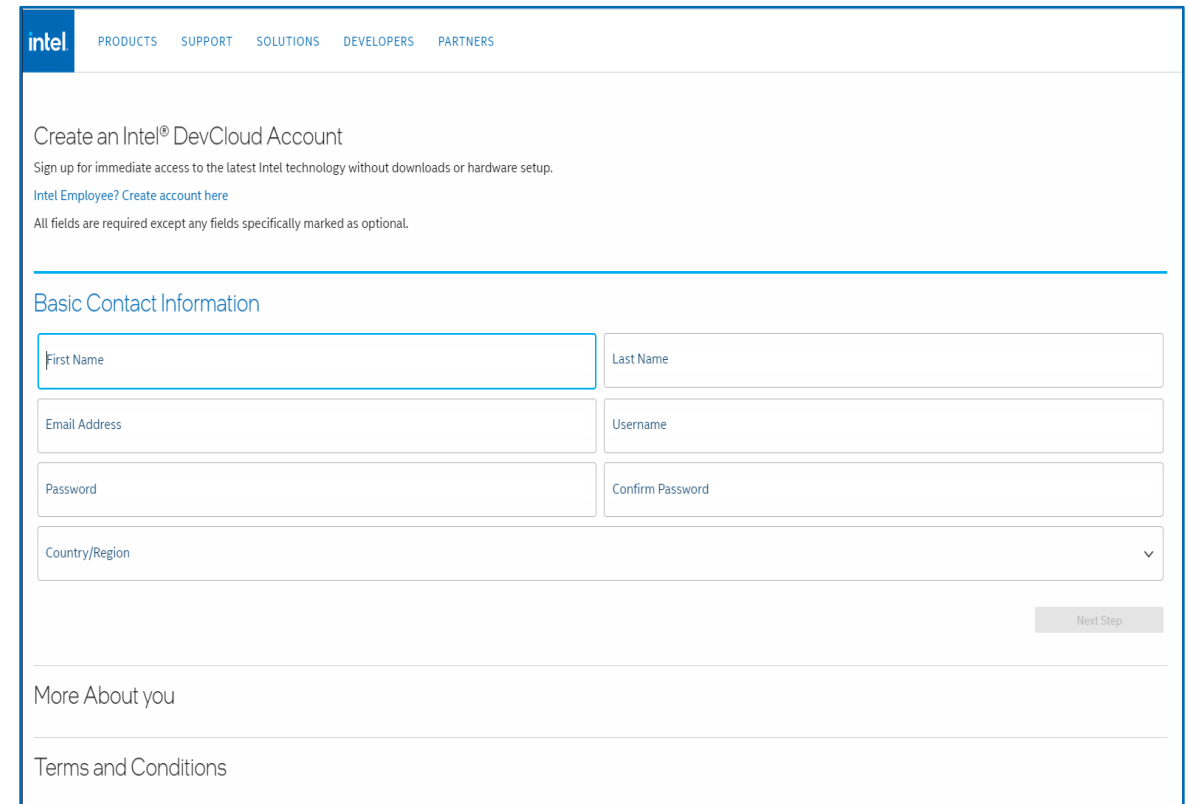
Introduction to ntel[®] DevCloud

Intel[®] DevCloud

- Free cloud environment for learning and project prototyping
- Complimentary access to a wide range of Intel[®] architectures (including FPGAs)
- Pre-installed Intel[®] optimized frameworks, tools, and libraries

Intel® DevCloud

- Sign up here:
 - <https://software.intel.com/devcloud>
 - Account for 120 days
 - Intel® oneAPI environment pre-installed and ready for use
 - Nodes with cards installed in the group fpga_runtime
 - Nodes with extra memory for full FPGA compiles in the group fpga_compile



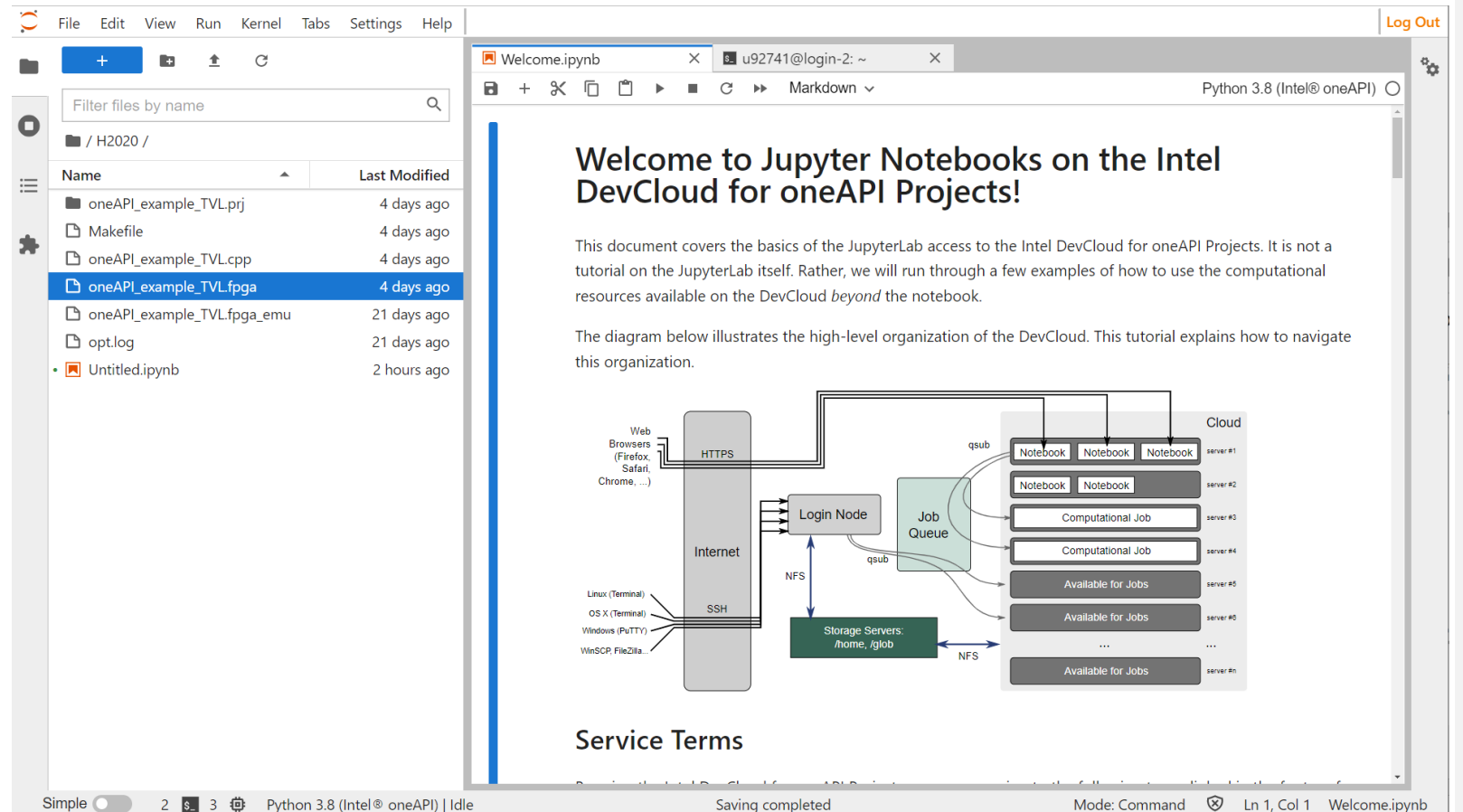
The screenshot shows the Intel DevCloud account creation page. At the top, there is a navigation bar with the Intel logo and links for PRODUCTS, SUPPORT, SOLUTIONS, DEVELOPERS, and PARTNERS. The main heading is "Create an Intel® DevCloud Account", followed by a sub-heading "Sign up for immediate access to the latest Intel technology without downloads or hardware setup." and a link "Intel Employee? Create account here". A note states "All fields are required except any fields specifically marked as optional." Below this is a section titled "Basic Contact Information" containing several input fields: First Name, Last Name, Email Address, Username, Password, and Confirm Password. There is also a dropdown menu for Country/Region. A "Next Step" button is located at the bottom right of the form. Below the form, there are links for "More About you" and "Terms and Conditions".

Use the Intel[®] DevCloud

- SSH to gateway
- Or Jupyter Notebooks via browser
- Job queue for compile and run
 - Job output into log file
- Access to single node also possible
- Session time limit:
 - default : 6hrs
 - max : 24hrs
- Many samples and tutorials in Git repo

<https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/DPC%2B%2BFPGA>

Jupyter Notebooks



The screenshot shows a Jupyter Notebook interface. On the left is a file explorer for the directory /H2020/. The file list includes:

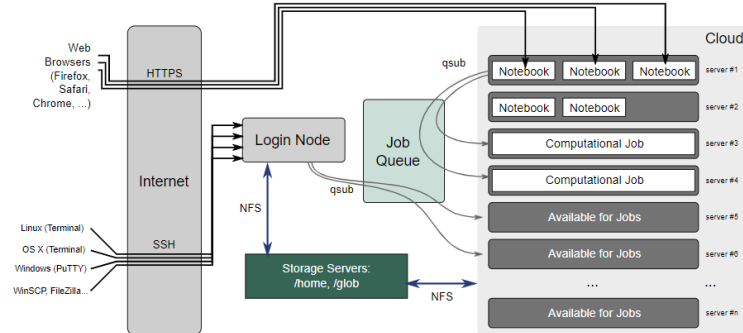
Name	Last Modified
oneAPI_example_TVL.prj	4 days ago
Makefile	4 days ago
oneAPI_example_TVL.cpp	4 days ago
oneAPI_example_TVL.fpga	4 days ago
oneAPI_example_TVL.fpga_emu	21 days ago
opt.log	21 days ago
Untitled.ipynb	2 hours ago

The main notebook cell contains the following text:

Welcome to Jupyter Notebooks on the Intel DevCloud for oneAPI Projects!

This document covers the basics of the JupyterLab access to the Intel DevCloud for oneAPI Projects. It is not a tutorial on the JupyterLab itself. Rather, we will run through a few examples of how to use the computational resources available on the DevCloud *beyond* the notebook.

The diagram below illustrates the high-level organization of the DevCloud. This tutorial explains how to navigate this organization.



The diagram illustrates the high-level organization of the DevCloud. It shows the flow of traffic from the Internet through a Login Node and a Job Queue to a Cloud of servers. The Cloud consists of multiple servers, some of which are running Jupyter Notebooks and others that are available for jobs. Storage Servers are also shown, connected to the Cloud via NFS.

Service Terms

Intel® DevCloud – Available FPGA Hardware

- What are you trying to use the Devcloud for?

- 1) Arria 10 PAC - RTL AFU, OpenCL
- 2) Arria 10 - OneAPI, OpenVINO
- 3) Stratix 10 - RTL AFU, OpenCL
- 4) Stratix 10 – OneAPI
- 5) Emulation
- 6) Compilation (bitstream creation)

Coming soon: Agilex cards with OneAPI support



Summary

FPGA compute acceleration with Intel® oneAPI

- Intel® oneAPI enables software engineers to use easily Intel FPGAs as compute accelerators without deep FPGA knowledge
- Unified programming model for different device types is making port between devices simple
- Modern programming language (DPC++)
- Support of industry standard debug and optimization tools like GDB, Intel® VTune™
- Including optimized RTL blocks as libraries
- Test easily in Intel® DevCloud ☺

- Let your use case also benefit from Intel® oneAPI FPGA compute acceleration

1
oneAPI



intel®